

Java™ Technology

Ivo Vondrak, Ph.D.

**Department of Computer Science
VŠB - Technical University of Ostrava**

ivo.vondrak@vsb.cz

<http://vondrak.cs.vsb.cz>

(JDK1.1 required...)

Java Technology - Ivo Vondrak '99



Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other



References

- ◆ **David Flanagan: Java in a Nutshell, O'Reilly & Associates, Inc., USA, 1996**
- ◆ **Sun Educational Services: Basic Java Programming, Sun Microsystems, USA, 1996**
- ◆ **Sun Educational Services: Advanced Java Programming, Sun Microsystems, USA, 1996**
- ◆ **Gary Cornell, Cay S. Horstmann: Core Java, The SunSoft Press, USA, 1996**
- ◆ **Sun Microsystems: The Java Development Kit, HTML document, Sun Microsystems, USA, 1999**
- ◆ **Mary Campione, Kathy Walrath: The Java Tutorial: Object Oriented Programming for the Internet, Addison-Wesley, USA, 1996**
- ◆ **Greg Voss: JavaBeans Tutorial, HTML document, Sun Microsystems, USA, 1997**
- ◆ **Peter B. Kessler, Roger Riggs: Remote Objects for Java, JavaOne, Sun's Worldwide Java Developer Conference, 1997**

Java as a Technology

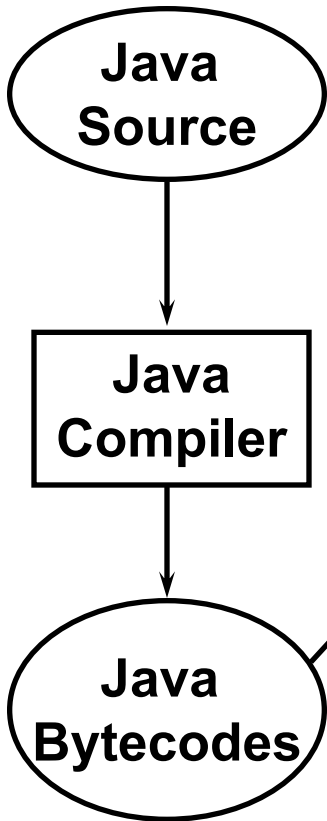
- ◆ **Architecture Neutral and Portable**
- ◆ **Object Oriented**
- ◆ **Robust, Dynamic and Secure**
- ◆ **Multithreaded**
- ◆ **Distributed**
- ◆ **Component Based Development (CBD) Support**

Architecture-Neutral

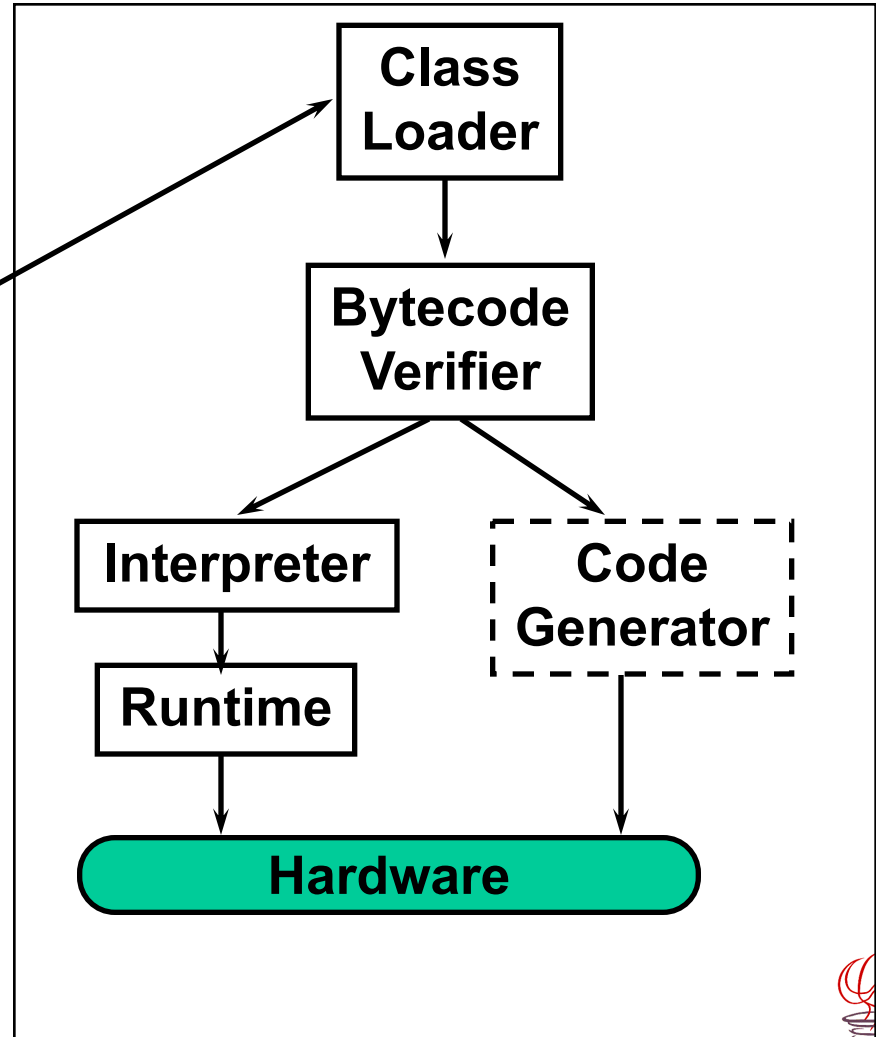
- ◆ Java source code is “compiled” into high-level, machine independent, *Java Bytecode* (.class files) format.
 - ◆ packages java.awt.*, java.net.*, java.applet.*
- ◆ *Java Virtual Machine* is an imaginary machine that is implemented by emulating it in software on a real machine.
 - ◆ JVM specification provides concrete definitions for implementation of instruction set, register set, class file format, stack ...

Compile Time and Runtime

Compile time



Runtime



Java Programming Structures

- ◆ **Basic Java Constructs**
- ◆ **Java Flow Control**
- ◆ **Object Concepts**
- ◆ **Exceptions and Exception Handling**
- ◆ **Types of Java Programs**
- ◆ **Packages**

Comments and Statements

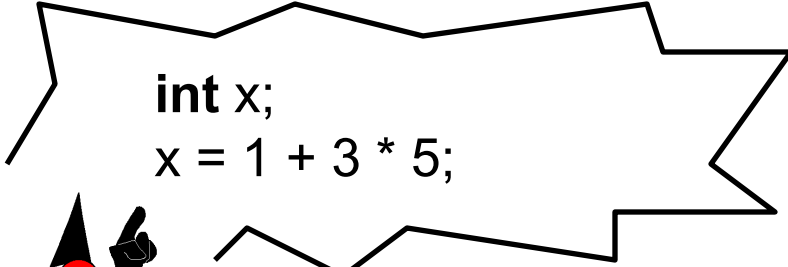
◆ Comments

// comment on one line

/* comment on one or more lines */

/** documenting comment, comment that should be included in any automatically generated documentation (the HTML files generated by the *javadoc* command */

◆ Statements form the smallest executable unit in a program



```
int x;  
x = 1 + 3 * 5;
```



Identifiers

- ◆ **Identifiers** name variables, functions, classes, and objects - anything that programmers need to identify and use. Identifiers start with letter, underscore or dollar sign and they are case-sensitive (e.g.):
 - ◆ ident
 - ◆ nameOfSomething
 - ◆ _name
 - ◆ User_name1
 - ◆ \$alsoValid

Keywords

abstract
boolean
break
byte
case
catch
char
class
continue
default
do
double
else
extends

int
interface
long
native
new
null
package
private
protected
public
return
short
static

Primitive Data Types

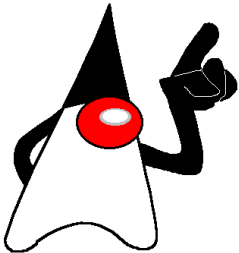
Java uses five basic element types: boolean, character, integer, floating point, and string.

Type	Contains	Default	Size
boolean	true or false	false	1 bits
char	unicode character	\u0000	16 bits
byte	signed integer	0	8 bits
short	signed integer	0	16 bits
int	signed integer	0	32 bits
long	signed integer	0	64 bits
float	floating point	0.0	32 bits
double	floating point	0.0	64 bits
String	string of chars	null	?? bits

Declarations and Assignments

```
int i, j;           // declare integer variables
long l = 100L;     // declare long variable
float x = 3.14159f; // declare and assign floating point
double y = 3.14159; // declare and assign double;
boolean cond;     // declare boolean variable
char c1, c2;      // declare char variables
String label;     // declare string variable

c1 = 'X';         // assign character
label = "Hello Duke"; // assign string
i = 1;           // assign integer variable
i = i+1;        // assign integer variable
```



Arrays

- ◆ Declaring arrays
- ◆ Creating arrays - arrays are created using the *new* keyword
- ◆ Using arrays

```
int[ ] x;                // or int x[ ];
int[ ] [ ] table;       // two dimensional array = array of arrays
x = new int[5];         // array of 5 integers created
table = new int[2] [10]; // two dimensional array 2x10 is created
String[ ] names = { "Hello" "Hi" "Good Morning" }
                        // array created with initial values

for (int i=0; i < x.length; i++) {
    x[i] = i+10;        // assign i+10 to element i of array x
}
```



Operators

- ◆ Java support almost all of the standard C operators.

= > < ! ~ ?:

== <= >= != && || ++ --

+ - * / & | ^ % << >> >>>

+= -= *= /= &= |= ^= %= <<= >>= >>>=

- ◆ Operator *instanceof* returns true if the object on the left-hand side is an instance of type specified on its right side.

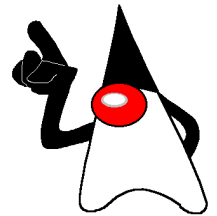
“Hello World!” instanceof String // returns true



Using Operators

- ◆ Operators for numbers behave as expected

```
int i = 1 + 3;           // i == 4
int j = 1;              // j == 1
j += 1;                 // j = j+1 => j == 2
i++;                    // i = i+1 => i == 5
boolean c1 = true;     // c1 == true
boolean c2 = !c1       // c2 == false
String name = "Richard" + "Gere";
```



- ◆ Casting - conversion between variable types

```
int i, j = 5;
float x = 10.2f;
i = (int) x / j;        // explicit cast needed, i == 2
i = (int) (x / (float) j);
```



Branching Statement *if-else*

The basic syntax:

```
if (boolean) {  
    statements;  
}  
else {  
    statements;  
}
```

```
float x, y;  
...  
if (y == 0) {  
    System.out.println("Divided by zero!")  
}  
else {  
    x = x / y;  
}
```



Branching Statement *switch*

The basic syntax:

```
switch (expr) {  
    case expr1:  
        statements;  
        break;  
    case expr2:  
        statements;  
        break;  
    default:  
        statements;  
}
```

```
int counter;  
...  
switch (counter % 3) {  
    case 0:  
        System.out.println("Hello");  
        break;  
    case 1:  
        System.out.println("Hi");  
        break;  
    case 2:  
        System.out.println("Bye");  
        break;  
}
```



Loop Statements *for*, *while*, and *do*

The basic syntax:

```
for (init_expr; test_expr; increment_expr) {  
    statements;  
}
```

```
while (boolean) {  
    statements;  
}
```

```
do {  
    statements;  
} while (boolean);
```

```
int i = 0;  
for (i=0; i < 10; i++) {  
    System.out.println("Value: "+ i);  
}  
int j = 0;  
while (j < 10) {  
    System.out.println("Value: "+ j);  
    j++;  
}  
int k = 0;  
do {  
    System.out.println("Value: "+ k);  
    k++;  
} while (k < 10);
```

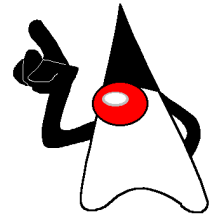


General Flow Control

- ◆ **break** [*label*]
- ◆ **continue** [*label*]
- ◆ **return** *expr*;
- ◆ **label: statement; // statement must be a loop statement**

```
loop: while (true) {  
    for (int i = 0; i < 100; i++) {  
        switch (c = in.read()) {  
            case -1:  
            case '\n': break loop;    // jumps out while  
            ...  
        }  
    }  
}
```

```
test: for (int i = 0; i < 100; i++) {  
    while (true) {  
        if (i > 10) continue test;    // jumps to next iteration of for  
    }  
    ...  
}
```



Object-Oriented Programming

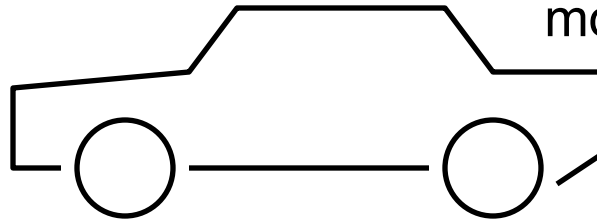
Object-oriented programming defines a program as a set of collaborating components (objects) with specified *behavior* and *state*.

- ◆ **Encapsulation** - a single object definition binds the operations and state particular to that object, and the implementation details are hidden.
- ◆ **Inheritance** - classes can be defined based upon existing class definitions for code reuse and enhancement.
- ◆ **Polymorphism** - The application of a function (method) to objects of different classes achieves the same semantic result.

Objects and Classes

- ◆ **Class** defines how an object will look - a template that defines the operation and behavior of an object.
- ◆ **Object** is an instance of a class - an example built from template.

Car operations:
drive, turn, stop

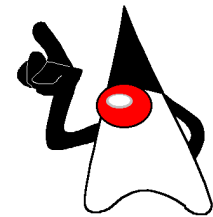


Car states:
moving, turning, stopped

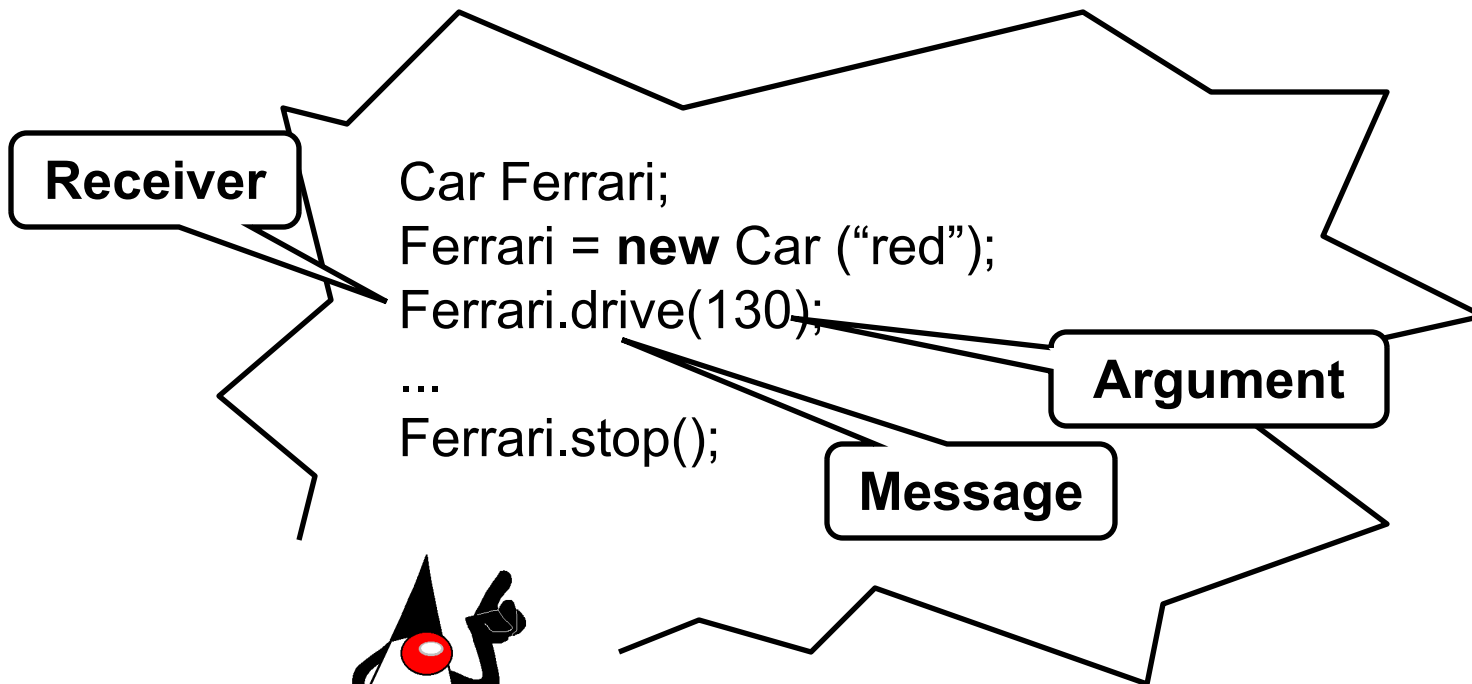
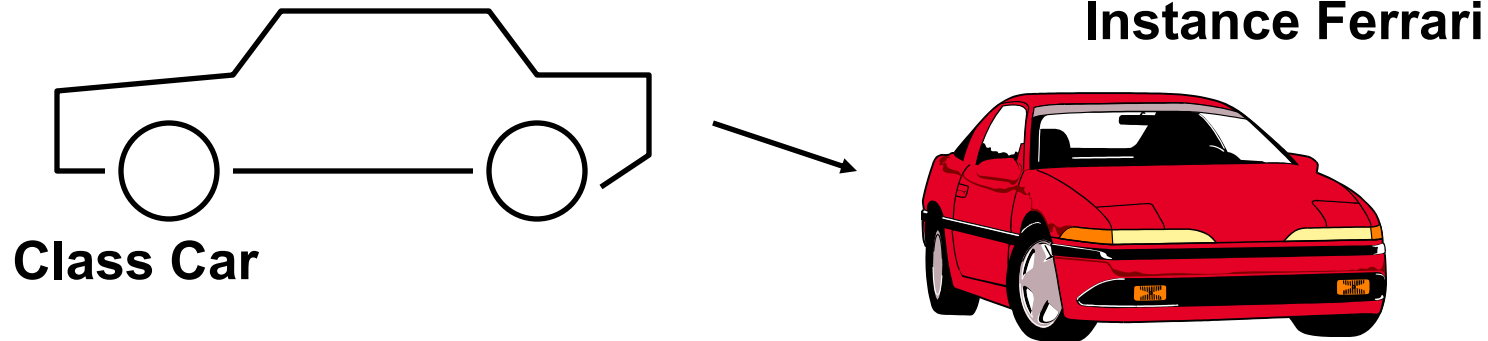
The class definition encapsulates all necessary attributes (state variables) and operations. The interface is consistent because all instances provide the same operations. Implementation details are hidden (the user do not have to know how the car stops, just applying the brakes will stop the car).

Car Class Definition

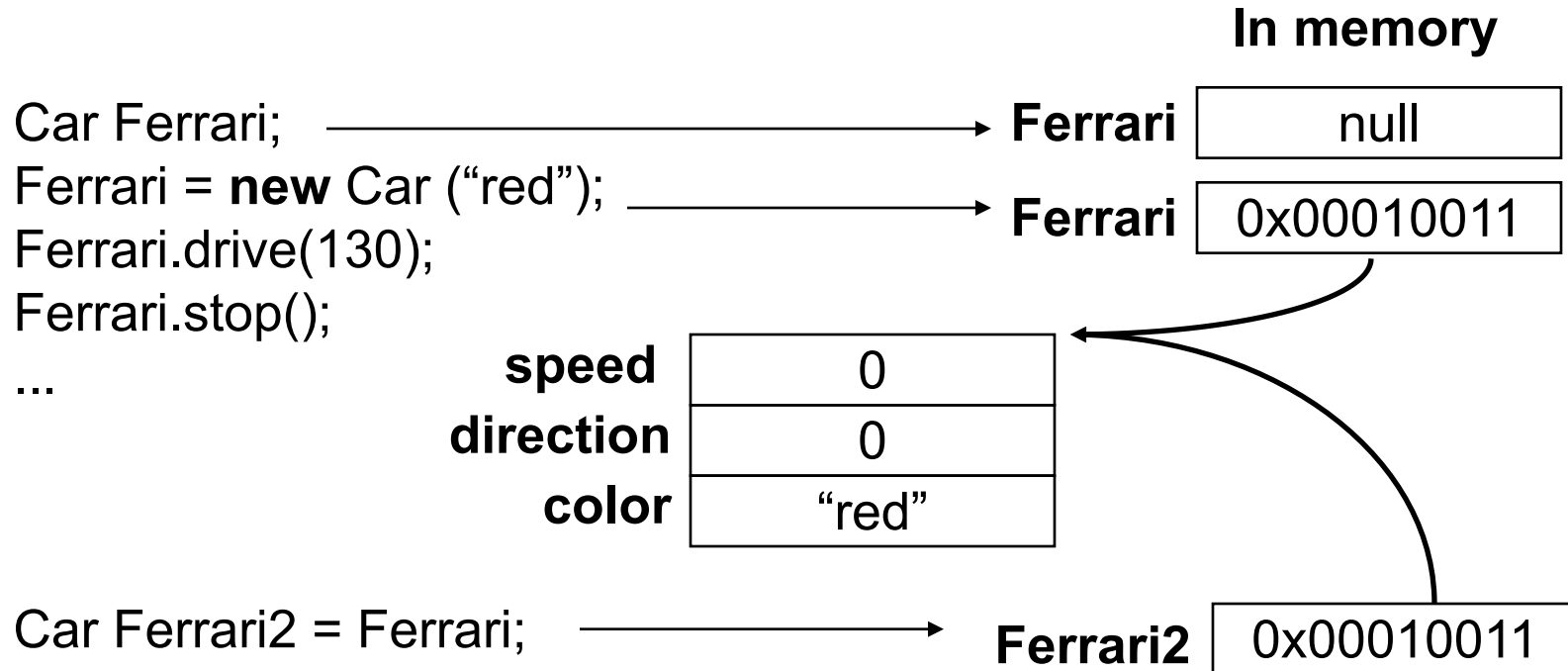
```
public class Car {  
    // State variables  
    private int speed, direction;  
    String color;  
  
    // Operations - methods  
    public Car (String color) {           // Constructor  
        this.color = color;  
    }  
    public void drive (int newSpeed) {  
        speed = newSpeed;  
    }  
    public void stop() {  
        speed = 0;  
    }  
    ...  
}
```



Creating and Using an Object

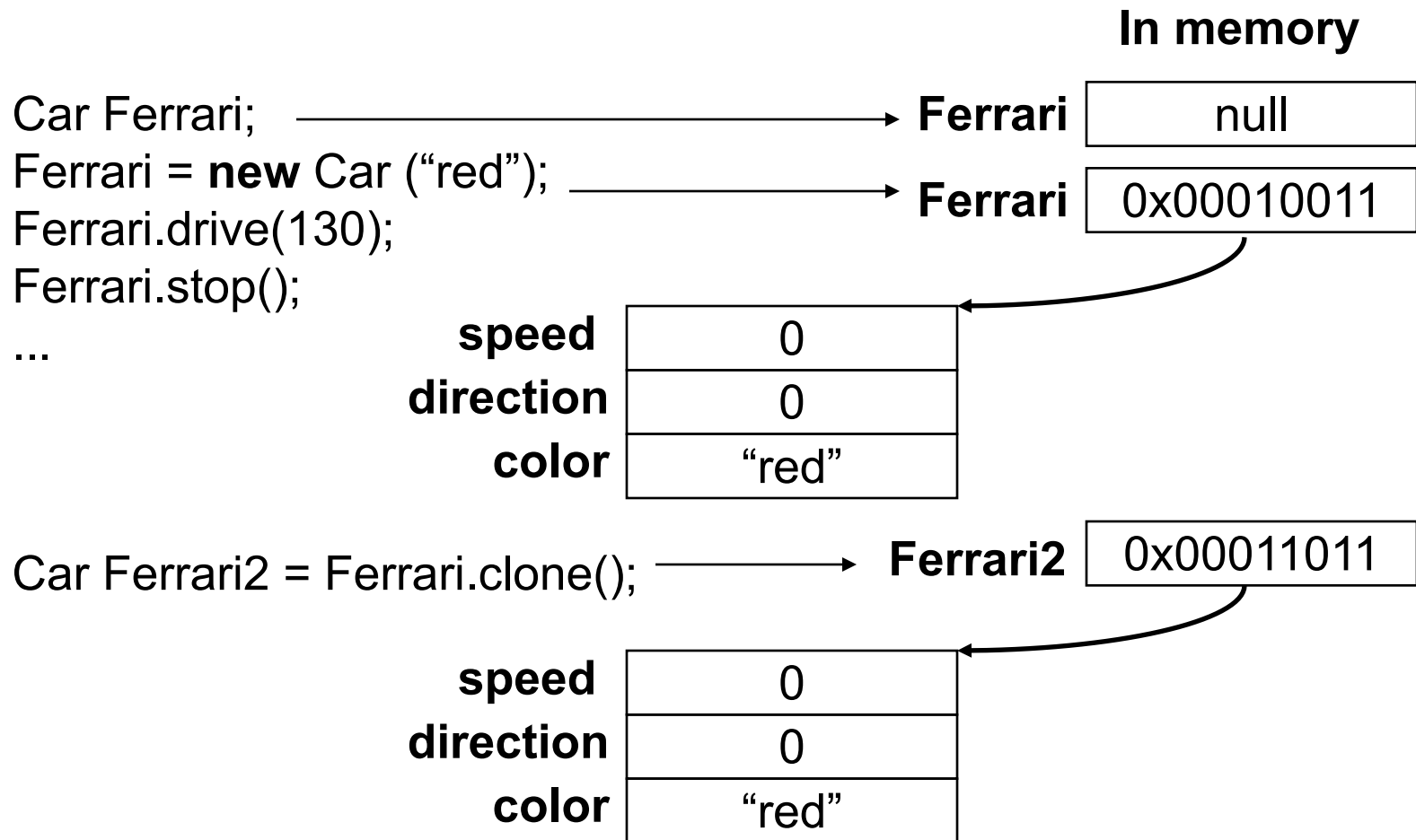


Memory Allocation



Copying Objects

To copy the object method *clone()* must be used.

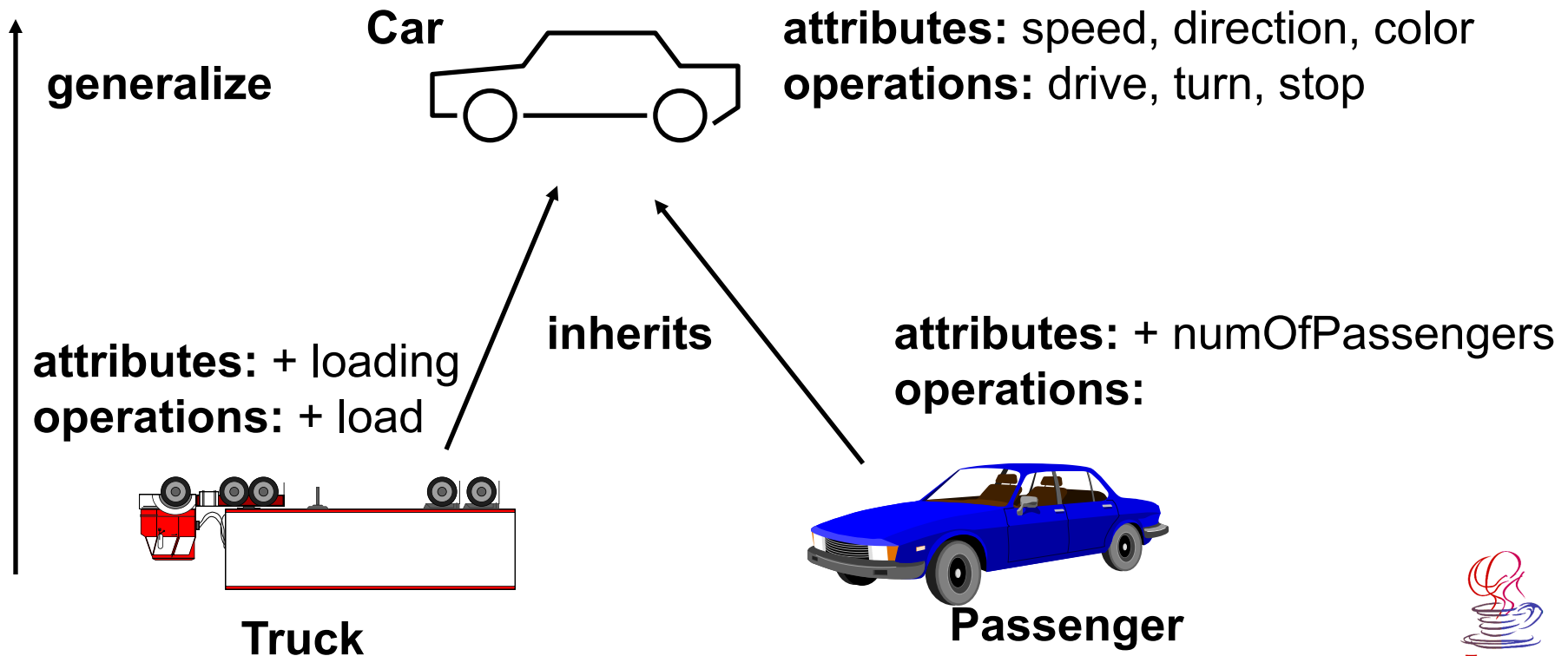


Checking Objects for Equality

- ◆ Operator `==` tests whether two variables refer to the same object (*identity*), not whether two object contain the same values.
- ◆ In Java, number of classes define an method `equals()` that compares containment of objects.

Generalization and Inheritance

- ◆ **Generalization** is the relationship between a class and one or more refined versions of it.
- ◆ **Inheritance** refers to the mechanism of sharing attributes and operations.



Subclassing

```
public class Truck extends Car {  
    // Additional state variables  
    private int loading;  
  
    // Operations - methods  
    public Truck (String color) {           // Constructor  
        super(color);  
    }  
    public void drive (int newSpeed) {     // Overriding of parent method  
        if (newSpeed <= 110)  
            super.drive(newSpeed);  
    }  
    public void load(int loading) {       // Additional method  
        this.loading = loading;  
    }  
}
```



Garbage Collection

- ◆ When an object is no longer being used, it should release its memory space.
- ◆ The collection and freeing of memory is the responsibility of a thread of code called *automatic garbage collector*.
- ◆ The garbage collector keeps track of all memory allocated with the *new* key keyword and also tracks who has access to that memory. When the access count reaches zero, the memory can be collected and freed.

Exceptions and Exception Handling

◆ Exception Handling

```
try {  
    critical_statements;  
}  
catch (ExceptionType e) {  
    // Handle exception object e  
}  
finally {  
    always_statements;  
}
```

```
int x, y;  
try {  
    x = 10 / y;  
}  
catch (Exception e) {  
    x = 1;           // Default value  
}
```



◆ Declaring Exceptions

```
void method(arg...) throws ExceptionType {...}
```

◆ Defining and Generating Exceptions

```
throw new MyException("text to show")
```

Types of Java Programs

- ◆ **Java applications**, stand-alone programs like any native programs.
- ◆ **Applets**, Java programs that are downloaded over WWW and executed by a Web browser
 - ◆ **Servlets**, server side components, which dynamically extend Java-enabled servers. They provide a general framework for services built using the request-response paradigm.

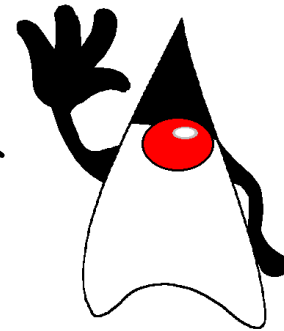
Java Application

```
public class HelloWorld {  
    public static void main(String[] arg) {  
        System.out.println ("Hello World!");  
    }  
}
```

*HelloWorld.java is compiled
and HelloWorld.class file is
produced.*

```
javac HelloWorld.java  
java HelloWorld
```

*Interpreter runs **main** from
HelloWorld.class.*



Java Applet

```
import java.awt.Graphics;
```

```
public class HelloWorldApplet extends java.applet.Applet {  
    // Display the content of the applet on the screen  
    public void paint(Graphics g) {  
        g.drawString("Hello World!",5,25);  
    }  
}
```



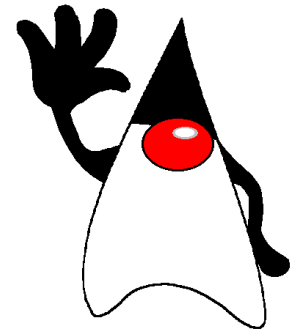
HTML Code for Applet

```
<HTML>  
<HEAD>  
<TITLE>Hello World Page</TITLE>  
</HEAD>  
<BODY>  
<P>  
My first applet says:  
<APPLET CODE="HelloWorldApplet.class" WIDTH=150 HEIGHT=25>  
</APPLET>  
</BODY>  
</HTML>
```

Program Structure

A program in Java consists of one or more class definitions, each of which has been compiled into its own *.class* file of Java Virtual Machine object code. In case of Java application one of these classes must define a method *main()*.

```
public class Echo {  
    public static void main(String[] arg) {  
        for (int i = 0; i < arg.length; i++)  
            System.out.print (arg[i] + " ");  
        System.out.println ("\n");  
    }  
}
```



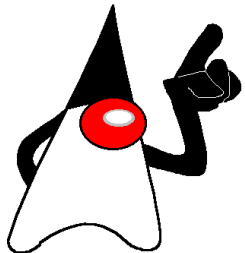
Packages and Classes

- ◆ Every compiled class is stored in a separate file (*.class*). This class must be stored in a directory that has the same components as the package name => *user.bank.Account* and *user\bank\Account.class*
- ◆ Source code file (*.java*) consists of one or more class definitions. Only one class may be declared *public* and the source file must have the same name.

Package and Import Statements

- ◆ The *package* statement must appear as the first statement. If omitted, the code is part of unnamed default package.
- ◆ The *import* statement makes Java classes available to the current class under an abbreviated name.

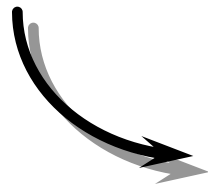
```
import user.bank.*;           // Abbreviate all class names from the package
import user.bank.Account;    // Only Account can be used abbreviated
...
new Account();               // Instead of new user.bank.Account()
```



The Java Class Path

- ◆ Java interpreter looks up classes relative to the directories specified by the **CLASSPATH** environment variable.

CLASSPATH=.;c:\jdk\lib\classes.zip;d:\java



.\user\bank\Account.class

or

d:\java\user\bank\Account.class

must exist, or the class is zipped in **classes.zip** including directory specification!

Java Class Library

Java class library (API) provides the set of classes that are guaranteed to be available in any Java environment. Those classes are in packages and they define one large hierarchy with one root - class *Object*. Programmer can define his/her new classes and packages that extend this hierarchy.

- ◆ **java.lang**: Classes that apply the language itself, which includes the *Object* class, the *String* class, and the *System* class.
- ◆ **java.util**: Utility classes, such as *Date*, as well as simple collection classes, such as *Vector* and *Hashtable*.
- ◆ **java.io**: Input and output classes for writing and reading from streams.
- ◆ **java.net**: Classes for networking support.
- ◆ **java.awt**: The Abstract Window Toolkit: Classes that implement a graphical user interface.
- ◆ **java.applet**: Classes to implement Java applets, including the Applet class itself.
- ◆ ...

Object-Oriented Approach

- ◆ **Object, Type, and Class**
- ◆ **Subtypes and Subclasses**
- ◆ **Creating and Destroying Objects**
- ◆ **Class Variables and Methods**
- ◆ **Data Hiding and Encapsulation**
- ◆ **Abstract Classes**

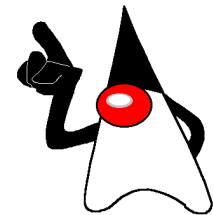
Object, Type, and Class

- ◆ **An Object is an identifiable individual entity with:**
 - ◆ **Identity:** a uniqueness which distinguishes it from all other objects
 - ◆ **Behavior:** services it provides in interactions with other objects
 - ◆ **Attributes:** data value held by an object
- ◆ **Type: visible interface and behavior**
 - ◆ Usually the object is a member of multiple types
 - ◆ Two objects with different implementation may be the same type
- ◆ **A Class is an abstraction of objects with similar implementation**
 - ◆ Every object is an instance of one class

Interface and Class Declaration

```
public interface CounterType {  
    public void increment();  
    public void decrement();  
}
```

```
public class Counter implements CounterType {  
    protected int value = 0;  
    public void increment() {  
        value++;  
    }  
    public void decrement() {  
        value--;  
    }  
    public void reset() {  
        value = 0;  
    }  
}
```



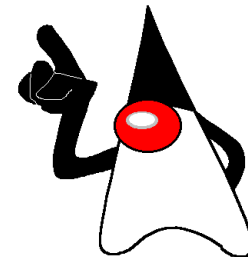
Subtypes and Subclasses

- ◆ **Extensions of the interface are described with subtypes**
 - ◆ Compatible services are still guaranteed
- ◆ **Re-use of implementation is provided by subclasses**
 - ◆ Each subclass can define its own implementation of attributes and services

Types and Subtypes

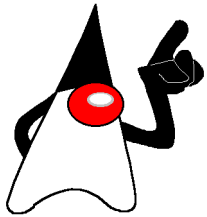
```
public interface BasicCounterType {  
    public void increment();  
    public void decrement();  
}
```

```
public interface CounterType extends BasicCounterType {  
    public void reset();  
}
```



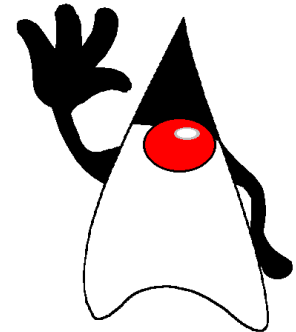
Class Implements Interface

```
public class IntCounter implements CounterType {  
    protected int value = 0;  
    public int getValue() {  
        return value;  
    }  
    public void increment() {  
        value++;  
    }  
    public void decrement() {  
        value--;  
    }  
    public void reset() {  
        value = 0;  
    }  
}
```



Main Application Class

```
public class CounterApp {  
    public static void main(String[ ] arg) {  
        CounterType counter = createCounter();  
        counter.reset();  
        counter.increment();  
        counter.increment();  
    }  
    ...  
    private static CounterType createCounter() {  
        return new IntCounter();  
    }  
}
```



What is the Benefit?

Re-use of the code for completely different implementations of Counter!

```
public class Stopwatch implements CounterType, TimeType {
    protected int hours, minutes, seconds;
    public String getTime() {
        return hours + ":" + minutes + ":" + seconds;
    }
    public void increment() {
        if (++seconds == 60) {
            seconds = 0;
            if (++minutes == 60) {
                minutes = 0;
                ++hours;
            }
        }
    }
    public void decrement() {...}
    public void reset() {hours = minutes = seconds = 0;}
}
```

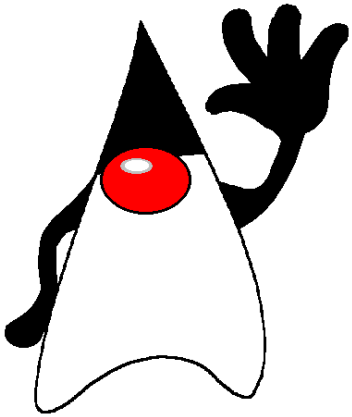


Two Implementations - One Type

```
public class StopwatchApp extends CounterApp {  
    public static void main(String[] arg) {  
        CounterType counter = createCounter();  
        counter.reset();  
        counter.increment();  
        counter.increment();  
    }  
}
```

Inherited!!!

```
private static CounterType createCounter() {  
    return new Stopwatch();  
}
```



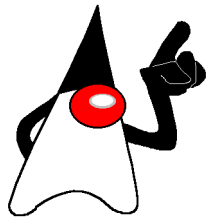
Subclasses and Inheritance

- ◆ The keyword *extends* is used to subclass an object.
- ◆ Every class has a superclass. If no superclass is specified with *extends* clause, the superclass is the class *Object* from `java.lang.*` package.
- ◆ Class declared with the *final* modifier cannot be subclassed.

Referring to Object Itself

The keyword *this* can be used to refer to an object itself. If no object reference is specified implicitly this is used.

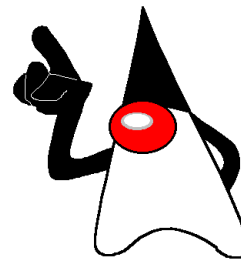
```
public class IntCounter implements CounterType {  
    protected int value = 0;  
    public void increment() {  
        this.value = this.value + 1;  
        // value = value+1 is perfectly valid as well  
    }  
    ...  
}
```



Referring to the Parent Class

The keyword *super* allows to reference methods that were overridden.

```
public class LimitedCounter extends IntCounter {  
    protected int limit = 100;  
    ...  
    public void increment() {  
        if (value < limit) {  
            super.increment(); // Calls increment() from IntCounter  
        }  
    }  
    ...  
}
```

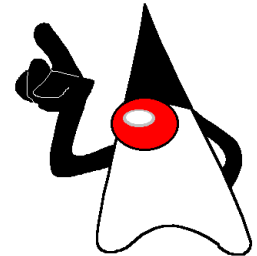


Constructors

- ◆ Every class has at least one constructor method responsible for initialization of the new object. If no constructor is defined Java creates *default* one with no arguments.
- ◆ The constructor name is always the same as the class name.
- ◆ The return object is implicitly an instance of the class. No return type is specified, nor is the *void* keyword used.

Multiple Constructors

```
public class IntCounter implements CounterType {  
    protected int value;  
    public IntCounter(int value) {  
        this.value = value;  
    }  
    public IntCounter() {  
        this(0); // The first constructor is invoked  
    }  
    ...  
}
```



Constructor Chaining

The keyword *super* can be used as the first method call in a constructor to call the parent's constructor.

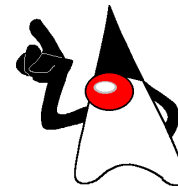
```
public class LimitedCounter extends IntCounter {  
    protected int limit;  
    public LimitedCounter(int limit) {  
        super(1);  
        this.limit = limit;  
    }  
    ...  
}
```



Object Destruction

- ◆ ***Garbage Collection*** destroys objects that are no longer needed. It runs as low priority thread when nothing else is going on or when the interpreter has run out of memory.
- ◆ ***Java finalizer*** method performs finalization for an object.

```
// Closes the stream when garbage is collected.  
// Checks the file descriptor fd first to make sure it is not already closed.  
protected void finalize() throws IOException {  
    if (fd != null) close();  
}
```



Class Variables and Methods

- ◆ **Class Variable** (*static*) - there is only one copy of this variable associated with the class and shared by all instances, e.g.

```
System.out.println("Hi there!");
```

- ◆ **Class Method** (*static*) - class method is invoked through class rather than through an instance. An implicit *this* reference is not passed!

```
Math.sqrt(12.34);
```

Data Hiding and Encapsulation

Modifier	public	default	protected	private
the same class	yes	yes	yes	yes
the subclass	yes	yes	yes	no
the same package	yes	yes	yes	no
anywhere	yes	no	no	no

Abstract Classes

- ◆ An *abstract* method has no body; it has a signature definition followed by a semicolon, e.g.

```
public abstract void foo();
```
- ◆ Any class with an abstract method is automatically abstract.
- ◆ An abstract class cannot be instantiated.
- ◆ A subclass of an abstract class can be instantiated if it overrides each of the abstract methods and provides an implementation.

Fundamental Techniques

- ◆ **Containers**
- ◆ **Indirect Invocation**
- ◆ **Input and Output Streams**
- ◆ **Threads and Multithreaded Programs**
- ◆ **Inner Classes**
- ◆ **Event Model**
- ◆ **Simple Networking**

Enumeration Type

This *interface* defines the methods necessary to enumerate, or iterate through set of values.

```
// package java.util
public interface Enumeration {
    public boolean hasMoreElements();
    public Object nextElement();
}
```



```
...
for (Enumeration e = vector.elements() ; e.hasMoreElements() ;) {
    System.out.println(e.nextElement());
}
```

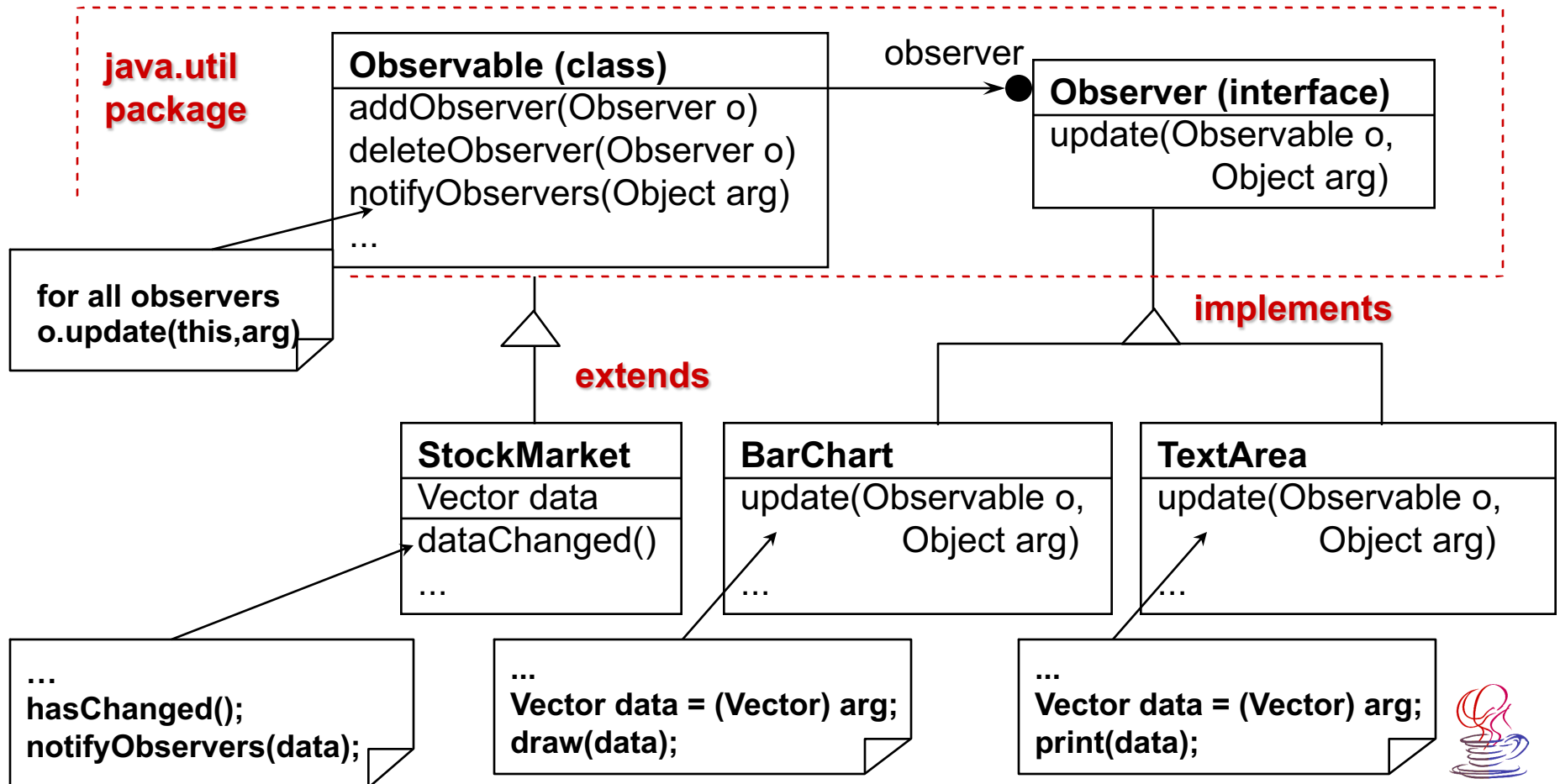


Containers

- ◆ The *Vector* class implements a growable array of heterogeneous objects.
- ◆ The *Stack* represents LIFO array of objects. The *Stack* extends *Vector* by implementing *push()*, *pop()*, *peek()* ...
- ◆ The *Dictionary* is the abstract parent for any class, such as *Hashtable*, which maps keys to values.

Observer Design Pattern

Intent - define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



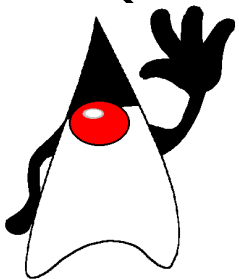
Input and Output Streams

- ◆ A *stream* is a flowing sequence of characters.
- ◆ A program can get input from a data source by reading a sequence characters from a stream attached to the source.
- ◆ A program can produce output by writing a sequence of characters to an output stream attached to a destination.
- ◆ Java development environment includes a package, *java.io*, that contains a set of input and output streams. The *InputStream* and *OutputStream* classes are the abstract superclasses that define the behavior for sequential input and output streams in Java.

Simple I/O Application

Intent - read a file and display its content on the standard output stream (screen).

```
import java.io.*;
public class ShowFile {
    public static void main(String[] arg) {
        try {
            File inputFile = new File(arg[0]);
            FileInputStream input = new FileInputStream(inputFile);
            int c;
            while ((c = input.read()) != -1)
                System.out.write(c); // System.out = PrintStream
            input.close();
        }
        catch (Exception e) {
            System.out.println("Error "+e);
        }
    }
}
```



Object Serialization

The capability to store and retrieve Java objects is essential to building all but the most transient applications. The key to storing and retrieving objects is representing the state of objects in a serialized form sufficient to reconstruct the object(s). Objects to be saved in the stream may support either the *Serializable* or the *Externalizable* Interface.

```
// Write objects
Counter counter = new Counter(0);
...
FileOutputStream f = new FileOutputStream("counter.obj");
ObjectOutput output = new ObjectOutputStream(f);
output.writeObject(counter);
output.writeObject(new Date());
```



```
// Read objects
FileInputStream f = new FileInputStream("counter.obj");
ObjectInput input = new ObjectInputStream(f);
Counter counter = (Counter) input.readObject();
Date date = (Date) input.readObject();
```



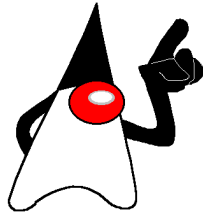
Threads

Thread represents a single process (sequence of statements) in execution on a system.

- ◆ The thread body consists entirely of the *run()* method and serves as a main routine for the thread.
- ◆ A thread can be in state **runnable**, **not runnable** (because of *suspend()*, *sleep()*, *wait()* or blocking I/O) and **dead** (because of *stop()* or completion of the *run()* method). Suspended thread can be activated by *resume()*.
- ◆ A thread can have a priority from *Thread.MIN_PRIORITY* (1) to *Thread.MAX_PRIORITY* (10).

Multithreaded Programs

Intent - create two threads that each print out their own text.



```
public class PrintThread extends Thread {
    String name;
    int delay;
    public PrintThread(String name, int delay) {
        this.name = name;
        this.delay = delay;
    }
    public void run() {
        try {
            sleep(delay);
        }
        catch (InterruptedException e) {}
        System.out.println("Hello from "+name);
    }
}
```

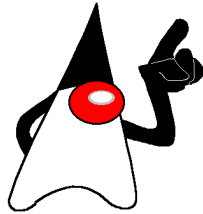
```
...
t1 = new PrintThread("#1", (int) (Math.random()*2000));
t2 = new PrintThread("#2", (int) (Math.random()*2000));
t1.start();
t2.start();
// start() calls run()
```



Interface Runnable

Interface Runnable declares a run() method.

```
public class Print implements Runnable {
    String name;
    int delay;
    public Print (String name, int delay) {
        this.name = name;
        this.delay = delay;
    }
    public void run() {
        try {
            Thread.sleep(delay);
        }
        catch (InterruptedException e) {}
        System.out.println("Hello from "+name);
    }
}
```



```
...
t1 = new Print ("#1", (int) (Math.random()*2000));
t2 = new Print ("#2", (int) (Math.random()*2000));
new Thread(t1).start();           // start() calls run()
new Thread(t2).start();
```



Synchronization

- ◆ Since Java is a multithreaded system, care must be taken to prevent multiple threads from modifying objects simultaneously. Section of code that must not be executed simultaneously are known as “critical section”.
- ◆ ***Statement synchronized:***
 - synchronized (expression) statement
 - expression must resolve to an object or array
 - statement is the code of critical section.

The synchronized statement attempts to acquire an exclusive lock for the object or array and it does not execute the critical section code until it can obtain this lock.
- ◆ ***Method modifier synchronized*** indicates that entire method is critical section code. For a synchronized instance method, Java obtains an exclusive lock on the class instance. For a synchronized class method, Java obtains an exclusive lock on the class.

Monitor

- ◆ A *monitor* is associated with a specific object (or array) and functions as a lock on that object. When a thread holds the monitor for some object, other threads are locked out and cannot inspect or modify this object.
- ◆ The Java runtime system allows a thread to re-acquire a monitor that it already holds because *Java monitors are reentrant*. Reentrant monitors are important because they eliminate the possibility of a single thread deadlocking itself on a monitor that it already holds.



```
class Reentrant {  
    public synchronized void a() {  
        b();  
        System.out.println("here I am, in a()");  
    }  
    public synchronized void b() {  
        System.out.println("here I am, in b()");  
    }  
}
```

Multiple-Thread Communication

- ◆ Method *wait()* of the Object class makes a thread wait until some condition occurs.
- ◆ Method *notify()* of the Object class tells a waiting thread that a condition occurred.

Example: Producer/Consumer

Intent - the *Producer* generates an integer between 0 and 9, stores it in a *Pool* object, and prints the generated number. To make the synchronization problem more interesting, the Producer sleeps for a random amount of time between 0 and 1000 milliseconds before repeating the number generating cycle. The *Consumer* consumes all integers from the Pool (the exact same object into which the Producer put the integers in the first place) as quickly as they become available.

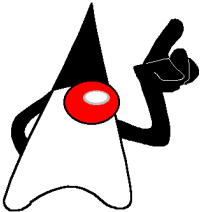
Producer

```
public class Producer extends Thread {  
    private Pool pool;  
    public Producer(Pool pool) {  
        this.pool = pool;  
    }  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            pool.put(i); // Wait until the previous value is consumed  
            System.out.println("Producer put: " + i);  
            try {  
                sleep((int)(Math.random() * 1000));  
            } catch (InterruptedException e) {}  
        }  
    }  
}
```



Consumer

```
public class Consumer extends Thread {  
    private Pool pool;  
    public Consumer(Pool pool) {  
        this.pool = pool;  
    }  
    public void run() {  
        int value;  
        for (int i = 0; i < 10; i++) {  
            value = pool.get(); // Wait until the value is produced  
            System.out.println("Consumer got: " + value);  
        }  
    }  
}
```



Shared Pool

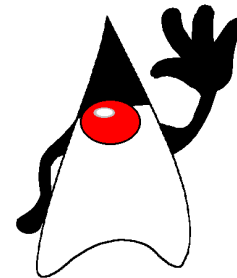
```
public class Pool {  
    private int contents;  
    private boolean isFull = false;  
    public synchronized int get() {  
        while (isFull == false) {  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        }  
        int value = contents;  
        isFull = false;  
        notifyAll();  
        return value;  
    }  
}
```



```
    public synchronized void put(int i) {  
        while (isFull == true) {  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        }  
        contents = i;  
        isFull = true;  
        notifyAll();  
    }  
}
```

Producer/Consumer Test

```
public class ProducerConsumerTest {  
    public static void main(String[] args) {  
        Pool pool = new Pool();  
        Producer p = new Producer(pool);  
        Consumer c = new Consumer(pool);  
        p.start();  
        c.start();  
    }  
}
```



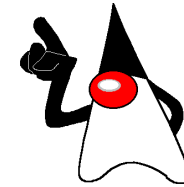
Inner Classes

- ◆ **Inner classes can be defined**
 - ◆ as members of other classes,
 - ◆ locally within a block of statements, or
 - ◆ (anonymously) within an expression.
- ◆ **The inner class's name is not usable outside its scope.**
- ◆ **The code of an inner class can use simple names from enclosing scopes, including both class and instance members of enclosing classes, and local variables of enclosing blocks.**
- ◆ **Inner classes have analogical purpose as C function pointers or Smalltalk blocks.**

Adapter Class

Intent - adapter class receives method invocations using a specified type interface on behalf of another object not of that type.

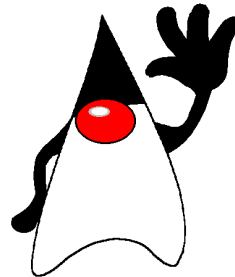
```
public class FixedStack {
    Object[] array = new Object[100];
    int top = 0;
    public void push(Object item) {
        array[top++] = item;
    }
    // other stack methods...
    class Enumerator implements Enumeration {
        int count = top;
        public boolean hasMoreElements() {
            return count > 0;
        }
        public Object nextElement() {
            if (count == 0) return null;
            return array[--count];
        }
    }
    public Enumeration elements() {
        return new Enumerator();
    }
}
```



Warning: Synchronization between stack and enumeration is missing!

Example: Fixed Stack

```
public class FixedStackTest {  
    public static void main(String[] arg) {  
        FixedStack s = new FixedStack();  
        s.push(new Integer(0));  
        s.push(new Float(3.14159));  
        for (Enumeration e=s.elements(); e.hasMoreElements();) {  
            System.out.println("Next element = "+e.nextElement());  
        }  
    }  
}
```



A Local Class

When a class definition is local to a block, it may access any names which are available to ordinary expressions within the same block.

```
public Enumeration enumerate(final Object array[]) {  
    class E implements Enumeration() {  
        int count = 0;  
        public boolean hasMoreElements()  
            { return count < array.length; }  
        public Object nextElement()  
            { return array[count++]; }  
    }  
    return new E();  
}
```

```
...  
Enumeration e =  
x.enumerate(array);  
Object o = e.nextElement();  
...
```

Anonymous Class

Intent - create “block of code” object that can be evaluated when needed.

```
public interface Block {  
    public void evaluate();  
    public void evaluateWith(Object o);  
}
```



```
public class Counter {  
    int value = 0;  
    // other counter methods ...  
    public Block display() {  
        return new Block() {  
            public void evaluate() {  
                System.out.println(value);  
            }  
            public void evaluateWith(Object o) {  
                ((PrintStream) o).println(value);  
            }  
        };  
    }  
}
```



```
...  
Block code = counter.display();  
if (stream == null)  
    code.evaluate();  
else  
    code.evaluateWith(stream);
```



Event Model

Events are mechanism for propagating of state change notifications between *source* object and one or more *listener* objects.

- ◆ Event notifications are propagated from sources to listeners by methods invocations on the target listener objects.
- ◆ Each distinct kind of event notification is defined as a distinct method. These methods are grouped in *EventListener* interfaces inherited from `java.util.EventListener`.
- ◆ Event listener classes identify themselves as interested in particular set of events by implementing some set of `EventListener` interfaces.
- ◆ The state associated with an event notification is normally encapsulated in an event state object that inherits from `java.util.EventObject` and which is passed as the sole argument to the event method.
- ◆ Event sources identify themselves as sourcing particular events by defining registration methods and accept references to instances of particular `EventListeners` interfaces.

Overview of Event Model

```
public synchronized addFooListener (FooListener fel) {  
    // register listener  
}
```



register
listener

Event Source

FooListener I;

fire
event

FooEvent

Event Listener

interface
reference

```
public class MyClass implements FooListener {  
    // MyClass implementation  
    public void fooEventOccurred(FooEvent e) {  
        //...  
    }  
}
```



Model/View Paradigm

Intent - implement dependency mechanism used in MVC.

```
public interface ModelChangeListener extends EventListener {  
    public void modelChanged(ModelEvent e);  
}
```



```
public class ModelEvent extends EventObject {  
    private Object argument;  
    public ModelEvent(Object source, Object argument) {  
        super(source);  
        this.argument = argument;  
    }  
    public Object getArgument() {  
        return argument;  
    }  
}
```



Model Definition

```
public class Model {
    private Vector listeners = new Vector();
    public synchronized void addModelChangeListener(ModelChangeListener l) {
        listeners.addElement(l);
    }
    public synchronized void removeModelChangeListener(ModelChangeListener l) {
        listeners.removeElement(l);
    }
    protected void notifyModelChanged(Object arg) {
        Vector l;
        ModelEvent e = new ModelEvent(this, arg);
        synchronized (this) {
            l = (Vector) listeners.clone();
        }
        for (int i=0; i < l.size(); i++) {
            ((ModelChangeListener) l.elementAt(i)).modelChanged(e);
        }
    }
}
```

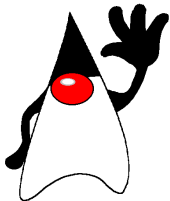


Example: Counter Model

```
public class Counter extends Model {  
    private int value = 0;  
    public void increment()  
        { setValue( getValue()+1 ); }  
    // similarly decrement ...  
    public void setValue(int value) {  
        this.value = value;  
        notifyModelChanged(new Integer(value));  
    }  
    public int getValue()  
        { return value; }  
}
```



```
public class ModelViewTest implements ModelChangeListener {  
    public static void main(String[] arg) {  
        ModelViewTest app = new ModelViewTest();  
        Counter counter = new Counter();  
        counter.addModelChangeListener(app);  
        counter.increment();  
        counter.decrement();  
    }  
    public void modelChanged(ModelEvent e) {  
        System.out.println(e.getArgument());  
    }  
}
```



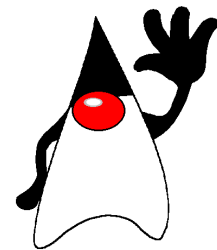
Simple Networking

- ◆ Loading applets from the network. Applets are referenced in a HTML file.
- ◆ Java programs can use URLs to connect to and retrieve information over the network. Uniform Resource Locator (URL) is an address of a resource on the Internet (*protocolID:resourceName*).
- ◆ Socket-based communication between programs. A socket is one end of a two-way communication link between two programs running on the network.
- ◆ Communication based on datagrams. The delivery of datagrams is not guaranteed nor is the order in which they are delivered.

Reading from URL

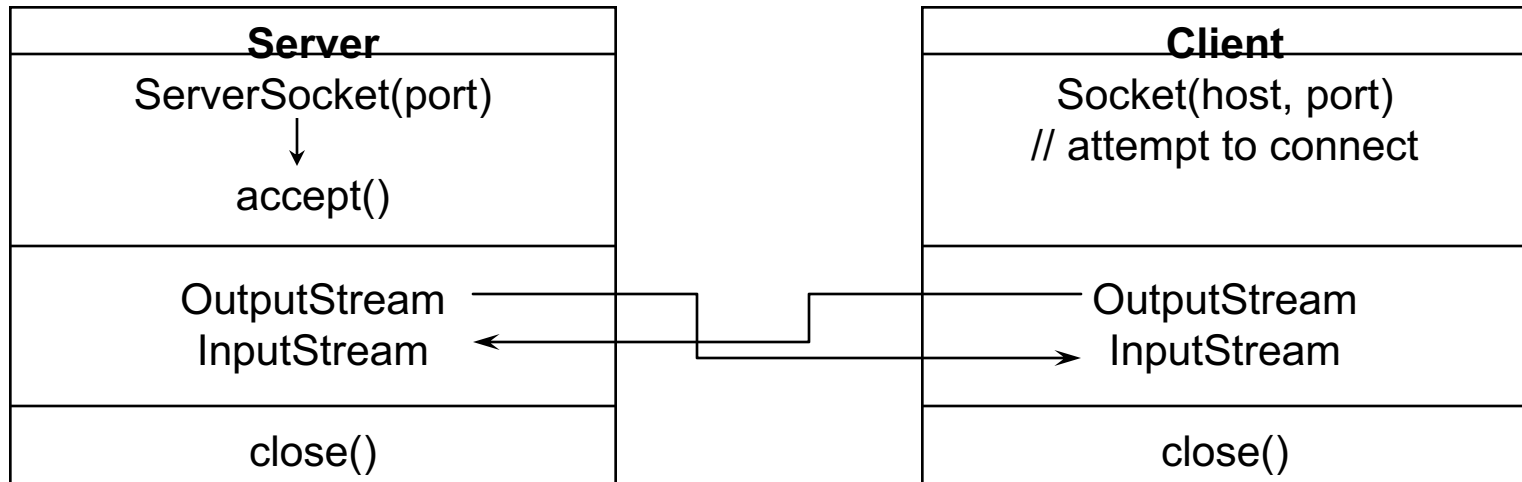
Intent - open URL `http://www.yahoo.com`, get an input stream on the connection, and read from the input stream.

```
import java.net.*;
import java.io.*;
public class ConnectionTest {
    public static void main(String[] args) {
        try {
            URL yahoo = new URL("http://www.yahoo.com/");
            URLConnection yahooConnection = yahoo.openConnection();
            DataInputStream dis = new DataInputStream(yahooConnection.getInputStream());
            String inputLine;
            while ((inputLine = dis.readLine()) != null)
                System.out.println(inputLine);
            dis.close();
        } catch (Exception e) {
            System.out.println("Error while communicating!");
        }
    }
}
```



The Socket Model

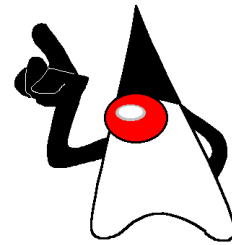
- ◆ The server establishes a port number and waits. When the client requests a connection, the server opens the socket connection with the `accept()` method.
- ◆ The client establishes a connection with host on a given port #.
- ◆ Both client and server communicate using `InputStream` and `OutputStream`.



Example: Simple Server

Intent - wait for client and send it a message when connected.

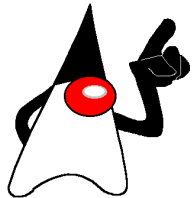
```
public class SimpleServer {
    ServerSocket server;
    String message = "Hello from server!";
    public void run() {
        try {
            server = new ServerSocket(5432, 5);
            listen();
        } catch (Exception e) { //... }
    }
    protected void listen() {
        try {
            while(true) {
                Socket client = server.accept();
                ObjectOutputStream out = new ObjectOutputStream(client.getOutputStream());
                out.writeObject(message);
                client.close();
            }
        } catch (Exception e) { //... }
    }
}
```



Example: Simple Client

Intent - connect to the server and get a message.

```
public class SimpleClient {  
    public void run(String host) {  
        try {  
            Socket server = new Socket(host, 5432);  
            ObjectInputStream input =  
                new ObjectInputStream(server.getInputStream());  
            System.out.println(input.readObject());  
            server.close();  
        }  
        catch (Exception e) {  
            System.out.println("Error while getting message!");  
        }  
    }  
}
```



Client/Server Application

```
public class ServerTest {  
    public static void main(String[] args) {  
        new SimpleServer().run();  
    }  
}
```



```
public class ClientTest {  
    public static void main(String[] args) {  
        new SimpleClient().run(args[0]);  
    }  
}
```

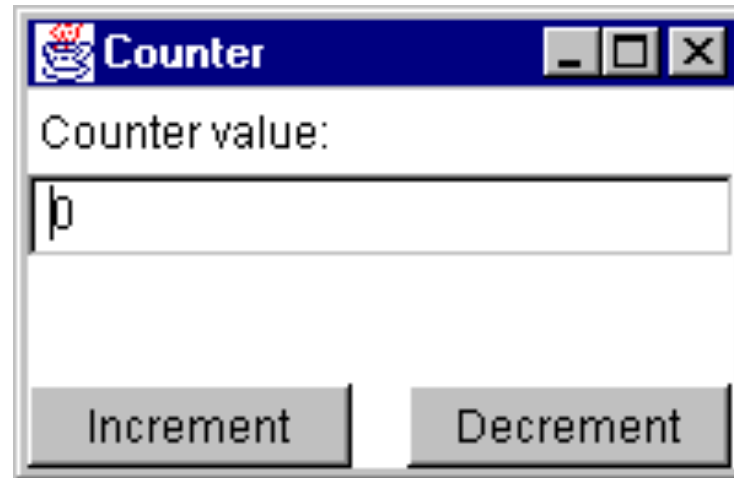


Graphical User Interface

- ◆ **Abstract Window Toolkit** - `java.awt.*`.
- ◆ **AWT Components** - containers (such as windows and menubars), leafs (such as buttons, lists, and textareas) and higher-level componets (such as file dialogs).
- ◆ **Layout Managers** - the way how to lay out components within containers.
- ◆ **AWT Event Model.**

Example: Counter with GUI

Intent - create counter with GUI. Counter will be controlled by Increment/Decrement buttons. Communication between counter and GUI will be based on *Observer/Observable pattern*.



Counter Type and Class

```
public interface CounterType {  
    public void increment();  
    public void decrement();  
}
```

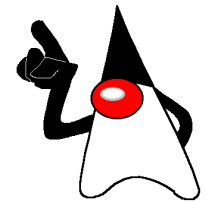


```
import java.util.*;  
public class Counter extends Observable implements CounterType {  
    private int value = 0;  
    public Counter(Observer o)  
        { addObserver(o); }  
    public void setValue(int value) {  
        this.value = value;  
        setChanged();  
        notifyObservers(new Integer(value));  
    }  
    public int getValue()  
        { return value; }  
    public void increment()  
        { setValue( getValue()+1 ); }  
    public void decrement()  
        { setValue( getValue()-1 ); }  
}
```



Counter GUI

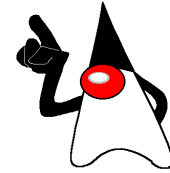
```
// Don't forget to import java.awt.*, java.awt.event.*, java.util.*
public class CounterGUI extends Frame implements Observer, ActionListener {
    protected Button inc = new Button(" Increment ");
    protected Button dec = new Button(" Decrement ");
    protected Label label = new Label(" Counter value: ");
    protected TextField value = new TextField("0");
    protected CounterType counter = new Counter(this);
    public CounterGUI() {
        setLayout(new BorderLayout(15,15));
        setTitle("Counter");
        Panel north = new Panel();
        north.setLayout(new BorderLayout());
        north.add("North",label);
        north.add("South",value);
        Panel south = new Panel();
        south.setLayout(new GridLayout(1,2,15,15));
        south.add(inc);
        south.add(dec);
        add("North",north);
        add("South",south);
        resize(200,130);
        inc.addActionListener(this);
        dec.addActionListener(this);
    }
}
```



Counter GUI: Event Handling

```
...  
// ActionListener's method implementation  
public void actionPerformed(ActionEvent e) {  
    if (e.getSource() == inc) {  
        counter.increment();  
    }  
    if (e.getSource() == dec) {  
        counter.decrement();  
    }  
}  
// Observer's method implementation  
public void update(Observable c, Object arg) {  
    value.setText(arg.toString());  
}  
}
```

```
public class CounterTest {  
    public static void main(String[] arg) {  
        new CounterGUI().show();  
    }  
}
```



Applets

- ◆ **The life cycle of an applet.**
- ◆ **Methods for drawing.**
- ◆ **User interface for applets.**
- ◆ **Security restrictions.**

Applet's Life Cycle

◆ Class Applet implements following life cycle methods:

- ◆ *init()* to initialize the applet each time it's loaded (or reloaded).
- ◆ *start()* to start the applet's execution, such as when the applet's loaded or when the user revisits a page that contains the applet.
- ◆ *stop()* to stop the applet's execution, such as when the user leaves the applet's page or quits the browser.
- ◆ *destroy()* to perform a final cleanup in preparation for unloading.

Example: Life Cycle

```
import java.applet.*;
import java.awt.*;
public class LifeCycle extends Applet {
    StringBuffer buffer;
    public void init() {
        buffer = new StringBuffer();
        display("Initializing...");
    }
    public void start()
        { display("Starting..."); }
    public void stop() {
        { display("Stopping..."); }
    }
    public void destroy()
        { display("Destroying..."); }
```

```
protected void display(String status) {
    buffer.append(status);
    showStatus(status);
    repaint();
}
public void paint(Graphics g)
    { g.drawString(buffer.toString(), 5, 15); }
```

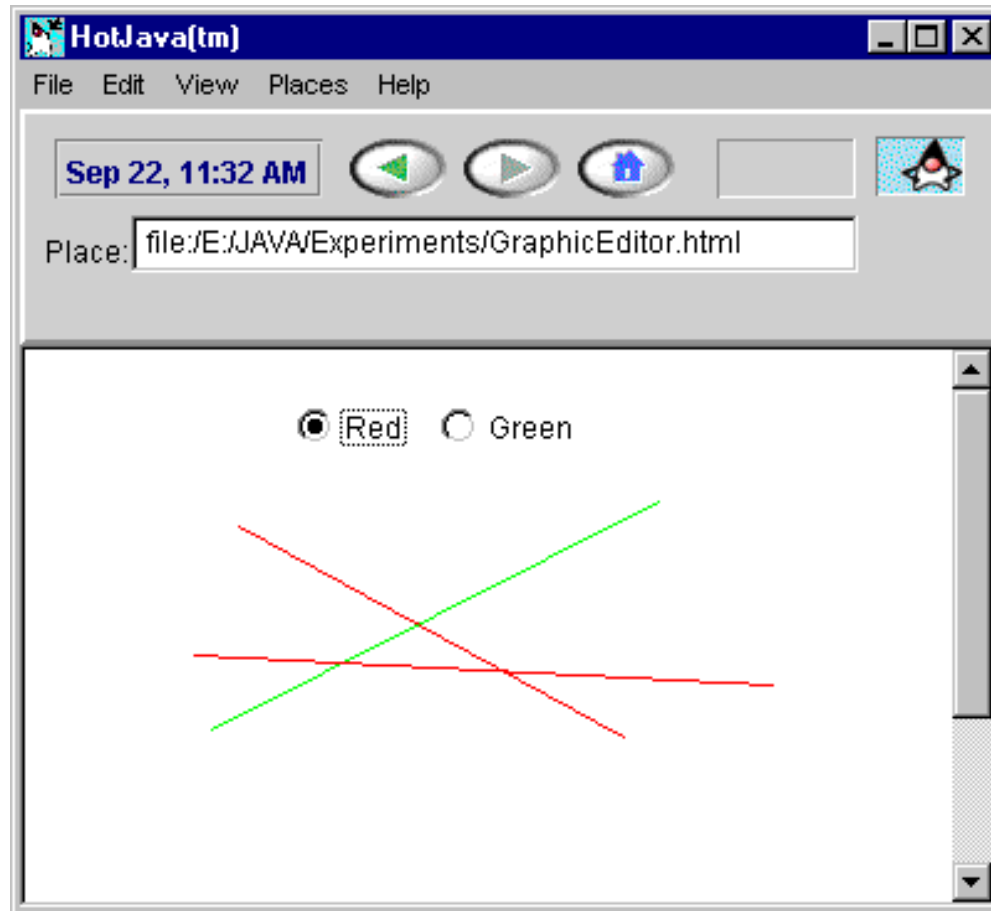
```
...
<applet code="LifeCycle.class" width=200 height=50> </applet>
```

Methods for Drawing

- ◆ ***paint(Graphics g)*** - this Applet's method is called when the applet drawing area must be refreshed. The derived class must override this method to draw anything different to the drawing area.
- ◆ ***repaint()*** - this method is called when the applet must be scheduled for repainting. This method should not be overridden.
- ◆ ***update(Graphics g)*** - this method is scheduled by *repaint*. The derived class can override following default behavior:
 - ◆ Setting the color to the background color.
 - ◆ Drawing a filled rectangle over the entire context.
 - ◆ Setting the color to the foreground color.
 - ◆ Calling then *paint(Graphics g)* method.

Example: Graphics Editor

Intent - create simple graphics editor for drawing lines in red or green color. To avoid flickering double-buffered drawing will be employed.



Applet Initialization

```
public class GraphicsEditor extends Applet  
    implements ItemListener, MouseListener, MouseMotionListener {
```

```
    Point p1,p2;
```

```
    Image buffer;
```

```
    Checkbox red, green;
```

```
    Color color;
```

```
    public void init() {
```

```
        CheckboxGroup cbg = new CheckboxGroup();
```

```
        red = new Checkbox("Red", cbg, true);
```

```
        green = new Checkbox("Green", cbg, false);
```

```
        Panel north = new Panel();
```

```
        north.add("West", red);
```

```
        north.add("East", green);
```

```
        add("North", north);
```

```
        buffer = createImage(300,300);
```

```
        color = Color.red;
```

```
        red.addItemListener(this);
```

```
        green.addItemListener(this);
```

```
        addMouseListener(this);
```

```
        addMouseMotionListener(this);
```

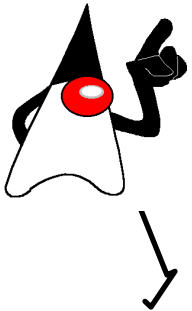
```
    }
```



Drawing Methods

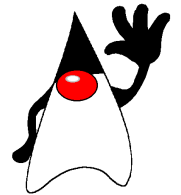
```
public void paint(Graphics g) {  
    g.drawImage(buffer,0,0,this);  
    if (p1 != null && p2 != null) {  
        g.setColor(color);  
        g.drawLine(p1.x, p1.y, p2.x, p2.y);  
    }  
}
```

```
// update is overridden to avoid cleaning of the drawing area  
public void update(Graphics g) {  
    paint(g);  
}
```



Event Handling

```
public void itemStateChanged(ItemEvent e) {
    if (e.getSource() == red) color = Color.red;
    if (e.getSource() == green) color = Color.green;
}
public void mouseClicked(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mousePressed(MouseEvent e) {
    p1 = e.getPoint();
}
public void mouseReleased(MouseEvent e) {
    Graphics g = buffer.getGraphics();
    g.setColor(color);
    g.drawLine(p1.x,p1.y,p2.x,p2.y);
    p1 = p2 = null;
    repaint();
}
public void mouseDragged(MouseEvent e) {
    p2 = e.getPoint();
    repaint();
}
public void mouseMoved(MouseEvent e) {}
}
```



Restrictions

- ◆ **Every browser implements security policies to keep applets from compromising system security.**
- ◆ **Current browsers impose the following restrictions on any applet that is loaded over the network:**
 - ◆ **An applet cannot load libraries or define native methods.**
 - ◆ **It cannot ordinarily read or write files on the host that's executing it.**
 - ◆ **It cannot make network connections except to the host that it came from.**
 - ◆ **It cannot start any program on the host that's executing it.**
 - ◆ **It cannot read certain system properties.**
 - ◆ **Windows that an applet brings up look different than windows that an application brings up.**

Advanced Techniques

- ◆ **JDBC** - database access from Java.
- ◆ **Java Reflection** - introspection about classes and object.
- ◆ **Java Beans** - component based development.
- ◆ **Remote Method Invocation (RMI)** - distributed objects in Java.

Database Access from Java

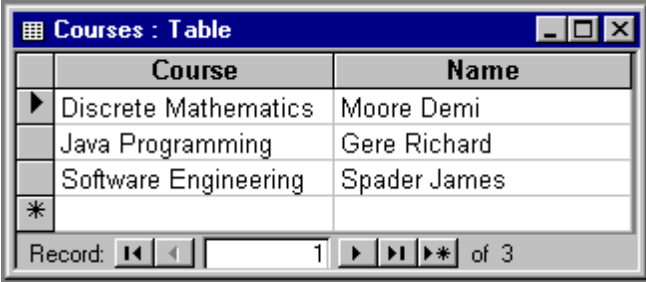
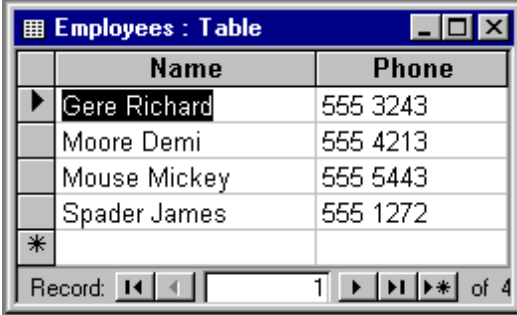
JDBC is a Java application programming interface (API) for executing SQL statements. It enables to send SQL statements to virtually any relational database.

- ◆ **JDBC makes it possible to do three things:**
 - ◆ establish a connection with a database
 - ◆ send SQL statements
 - ◆ process the results.

Example: Java DB Access

Intent - create Java application that will display information about courses, responsible lecturers and their phone numbers. The information is stored in the database *Education* that consists of two relational tables.

Education

Courses		Employees	
			

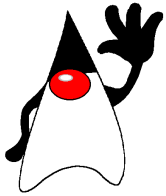
**SELECT Employees.Name, Employees.Phone, Courses.Course
FROM Employees, Courses WHERE Employees.Name = Courses.Name;**



List Courses : Select Query			
	Name	Phone	Course
▶	Spader James	555 1272	Software Engineering
	Gere Richard	555 3243	Java Programming
	Moore Demi	555 4213	Discrete Mathematics
*			
Record: 1 of 3			

JDBC™ Application

```
import java.sql.*;
public class JDBCTest {
    public static void main(String[] arg) {
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection con = DriverManager.getConnection("jdbc:odbc:Education");
            Statement statement = con.createStatement();
            ResultSet rs = statement.executeQuery(
                "SELECT Employees.Name, Employees.Phone, Courses.Course
                FROM Employees, Courses
                WHERE Employees.Name = Courses.Name;");
            while (rs.next())
                System.out.println(rs.getString("Name")+ " ("
                    +rs.getString("Phone")+"): "+rs.getString("Course"));
        } catch (Exception e) {}
    }
}
```



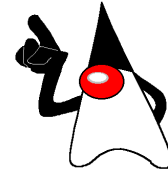
Java Reflection

- ◆ **Java reflection tools enable introspection about the classes and objects in the current JVM:**
 - ◆ A Field object represents a reflected field (a class variable or an instance variable).
 - ◆ A Method object represents a reflected method (an abstract method, an instance method, or a class method).
 - ◆ A Constructor object represents a reflected constructor.

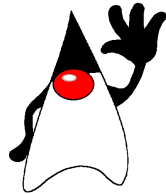
Example: Introspection

Intent - introspect class of an unknown object, find its display method and invoke it.

```
public class Unknown {  
    public void display() {  
        System.out.println("The display method invoked!");  
    }  
    public void method1() {}  
    public void method2() {}  
}
```



```
public class ReflectionTest {  
    public static void main(String[] arg) {  
        Unknown obj = new Unknown();  
        Class cl = obj.getClass();  
        Method[] methods = cl.getMethods();  
        for (int i=0; i < methods.length; i++) {  
            if (methods[i].getName().equals("display"))  
                try { methods[i].invoke(obj, null); }  
                catch (Exception e) {}  
        }  
    }  
}
```



Component Based Development

A JavaBean™ is a reusable software component that can be manipulated visually in a builder tool.

- ◆ What differentiates bean from a class instance is a possibility to inform about its behavior and properties during *design-time* in contrast to *run-time*.
- ◆ Typical Java Beans features are following:
 - ◆ Support for **introspection** allowing a builder tool to analyze how a bean works.
 - ◆ Support for **customization** allowing a user to alter the appearance and behavior of a bean.
 - ◆ Support for **events** allowing beans to fire events, and informing builder tools about both the *events they can fire* and the *events they can handle*.
 - ◆ Support for **properties**, both for customization and for programmatic use.
 - ◆ Support for **persistence**, so that a bean can be customized in an application builder and then have its customized state saved and restored later.

Scenario of App[let/lication] Building

- ◆ ***Loading*** the development environment (standard or visual).
- ◆ ***Laying out*** the app[let/lication] by adding beans (drag&drop or programmatically).
- ◆ ***Customizing*** the beans - properties of involved beans are changed.
- ◆ ***Wiring*** the beans based on *event->action* (method) paradigm.
- ◆ ***Packaging*** the app[let/lication] by generating .jar file containing beans' .class and .ser (serialized objects) files.

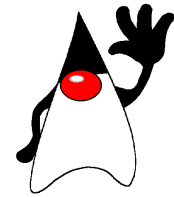
Example: Counter Bean

```
import java.beans.*;
import java.io.*;
public class CounterBean implements Serializable {
    protected int value = 0;
    private PropertyChangeSupport listeners = new PropertyChangeSupport(this);
    public synchronized int getValue()
        { return value; }
    public synchronized void setValue(int newValue) {
        int oldValue = value;
        value = newValue;
        listeners.firePropertyChange("value", new Integer(oldValue), new Integer(newValue));
    }
    public synchronized void increment()
        { setValue(getValue() + 1); }
    public synchronized void decrement()
        { setValue(getValue() - 1); }
    public void addPropertyChangeListener(PropertyChangeListener l)
        { listeners.addPropertyChangeListener(l); }
    }
    public void removePropertyChangeListener(PropertyChangeListener l)
        { listeners.removePropertyChangeListener(l); }
    }
```



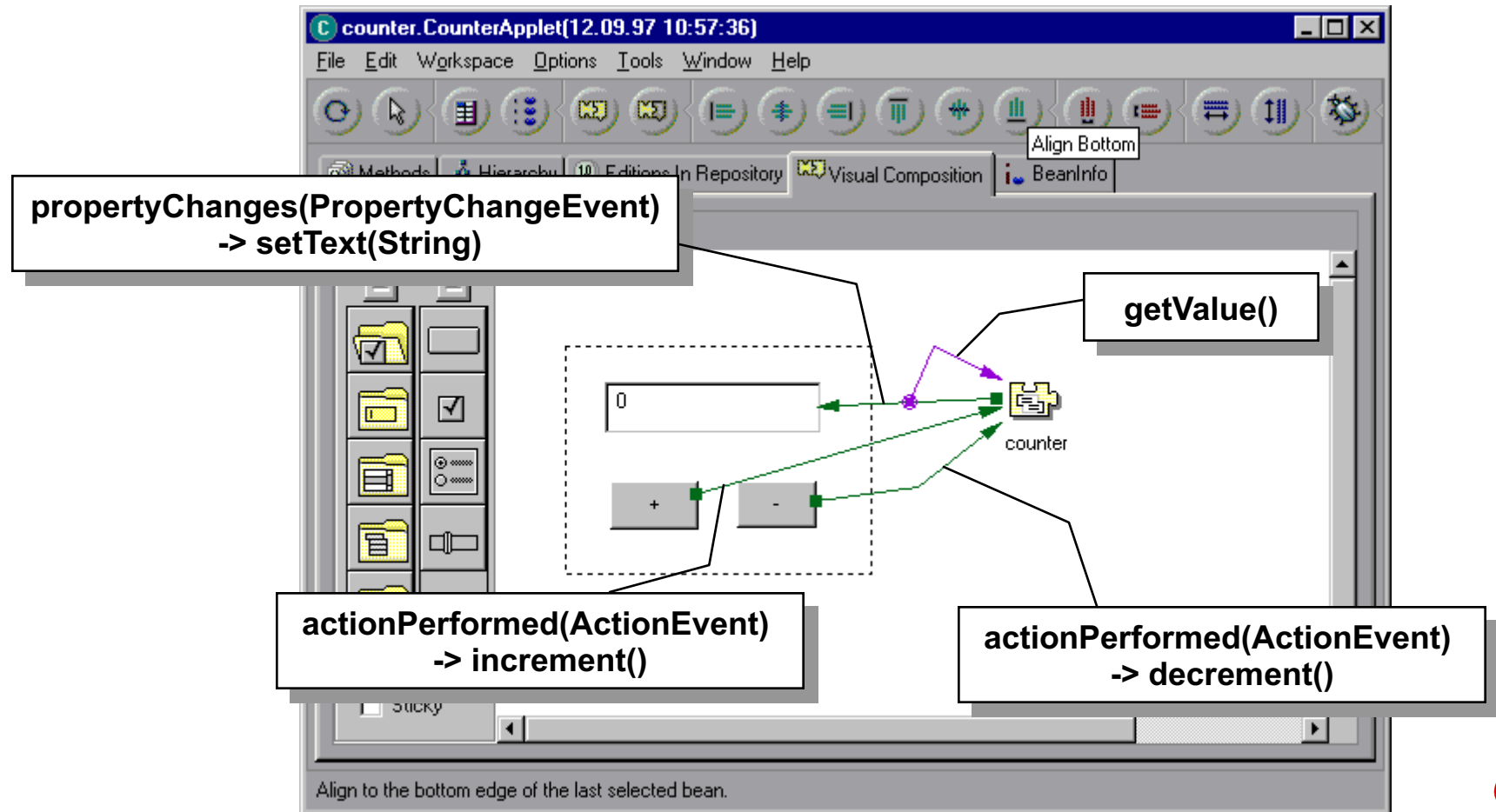
Counter Bean Applet

```
public class CounterBeanApplet extends Applet
    implements ActionListener, PropertyChangeListener {
    // Definition of GUI elements - buttons inc and dec, text field value, etc...
    protected CounterBean counter;
    public void init() {
        try {
            ClassLoader cl = this.getClass().getClassLoader();
            counter = (CounterBean) Beans.instantiate(cl, "CounterBean");
            // counter = new CounterBean(); can be used in case that there is no persistence info
            counter.addPropertyChangeListener(this);
            // Laying out of GUI components ...
        }
        catch (Exception e) {}
    }
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == inc) counter.increment();
        if (e.getSource() == dec) counter.decrement();
    }
    public void propertyChange(PropertyChangeEvent ev) {
        value.setText(ev.getNewValue().toString());
        // value.setText(new Integer(counter.getValue()).toString());
        // is perfectly valid, too.
    }
}
```



Visual Building

Intent - connect the counter component with an applet in a visual builder tool.

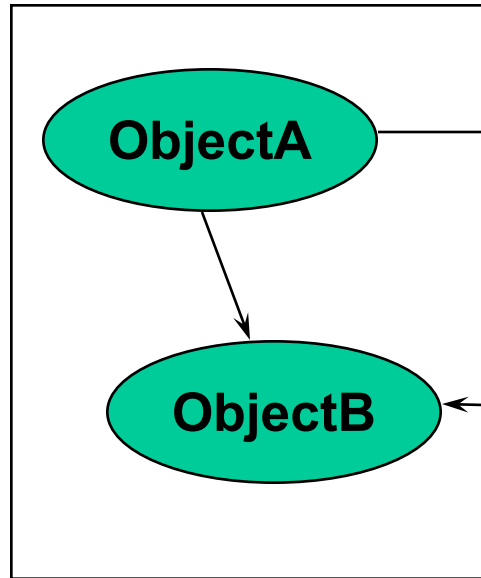


Remote Objects

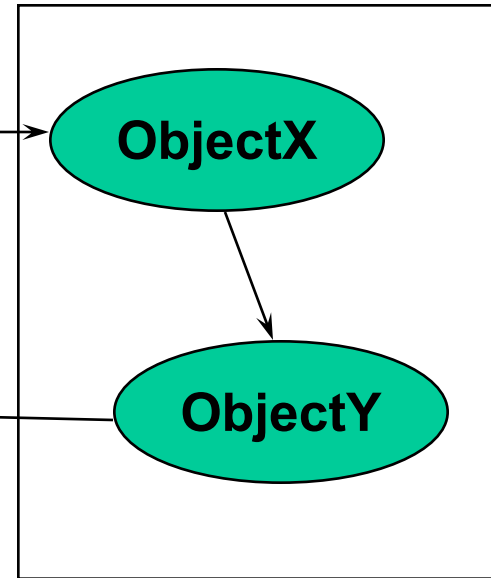
- ◆ **Remote objects** are objects whose methods can be invoked from another Java VM.
- ◆ **Remote interface** is Java interface that declares the methods of remote object.
 - ◆ Remote object may support many remote interfaces.

Distribution of Objects

App[lication|let] 1



App[lication|let] N



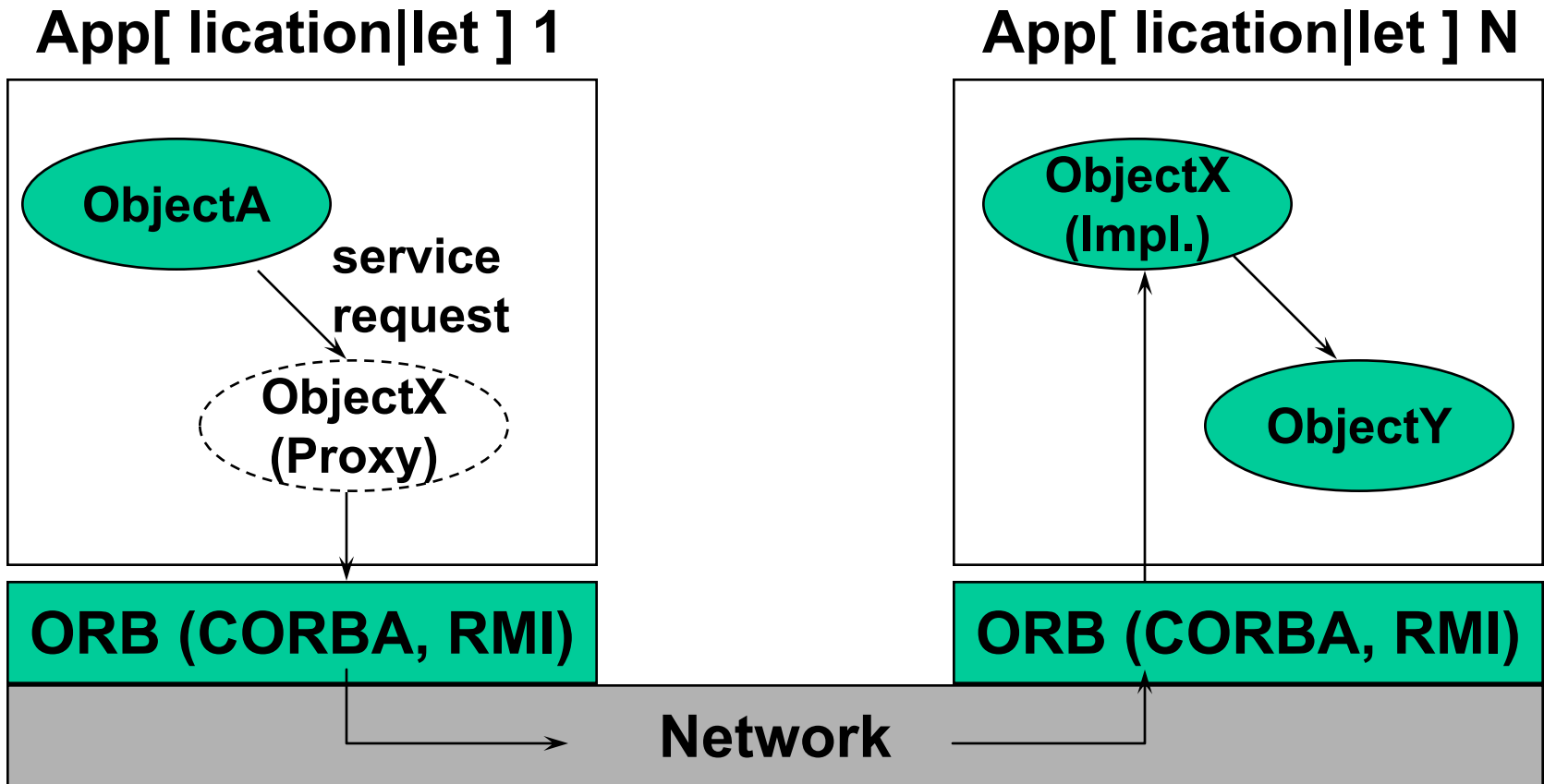
service
request

ObjectB

ObjectY

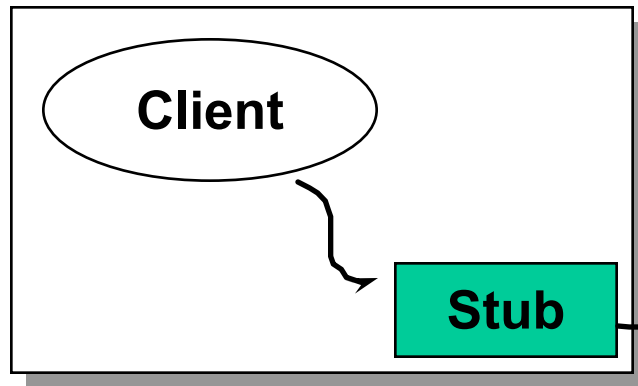
Network

Object Request Broker



Proxy represents the remote object in communication with client.

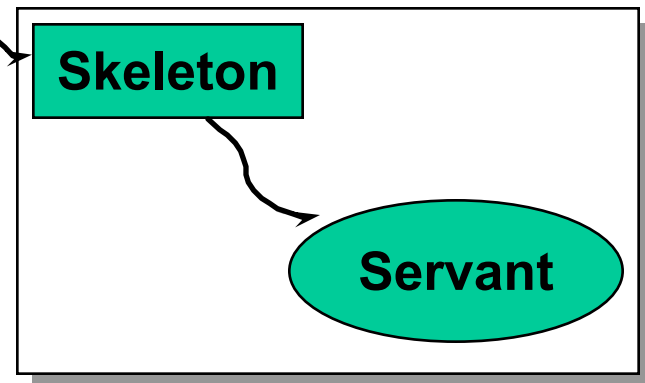
Stubs and Skeletons



Client operates on a stub (remote object client-side proxy) as local object. Stub marshals arguments for transmission to server.

Skeleton unmarshals arguments and calls servant.

Servant receives call from skeleton as local call.



And the process is reversed for returning results.

Java IDL and Java RMI

- ◆ **Java IDL (Interface Definition Language) is heterogeneous solution.**

 - ◆ Uses a standard, language-neutral, interface description language.
 - ◆ Used open, standard protocols to interact with services written in many languages (CORBA).

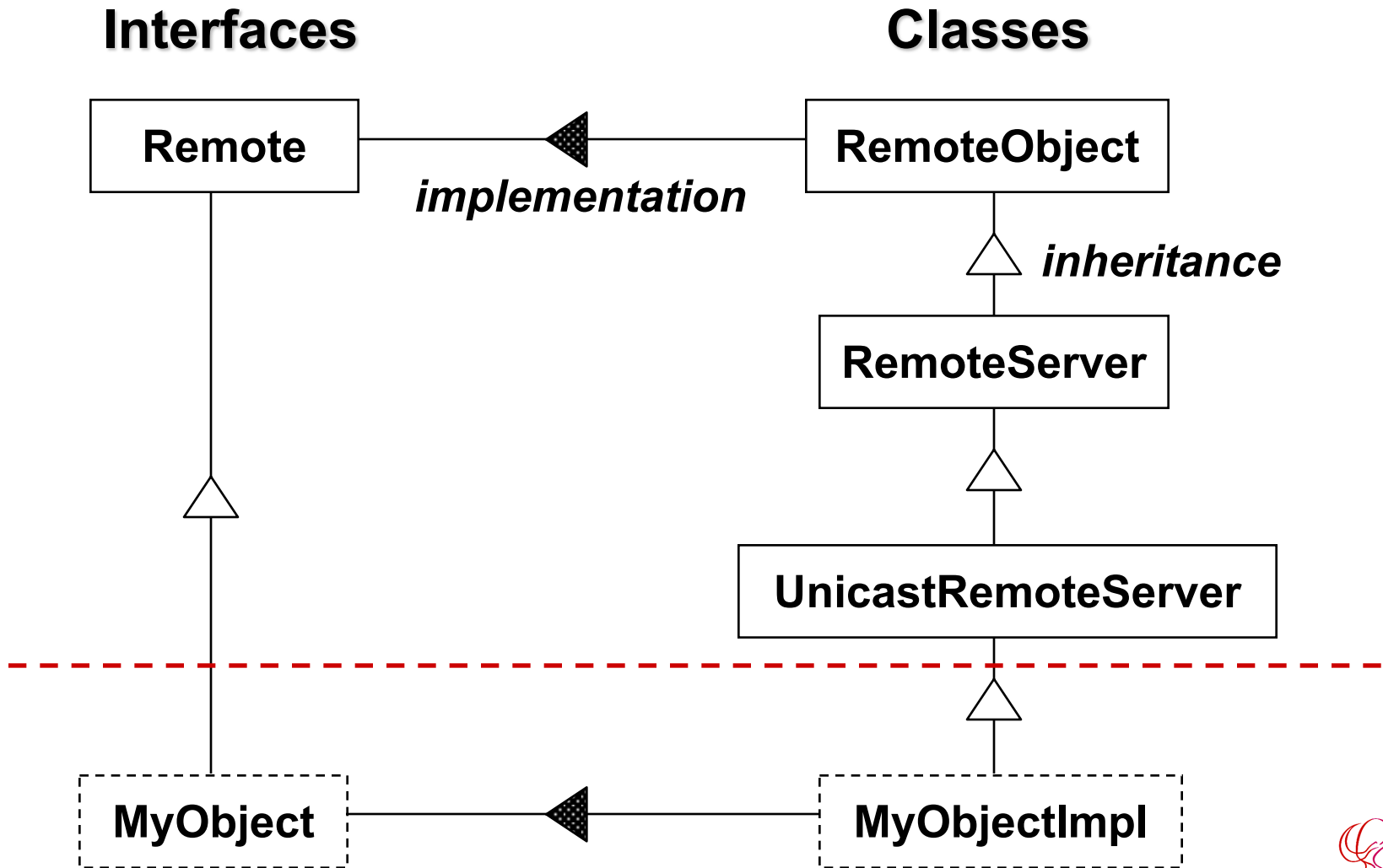
- ◆ **Java RMI (Remote Method Invocation) is a Java-only solution.**

 - ◆ Uses Java interface and data types to describe remote interfaces.
 - ◆ Uses specialized protocols to interact with objects written in Java.

Remote Method Invocation

- ◆ **Method invocation between objects in different Java VM.**
- ◆ **Pure Java interfaces - no new interface definition language is needed.**
- ◆ **Pass and return and Java Object.**
- ◆ **Dynamic loading of classes.**

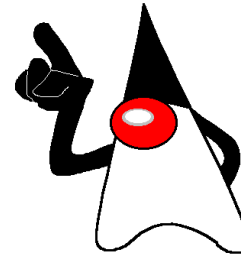
RMI Interfaces and Classes



Example: Remote Counter

Intent - create counter as a remote object that can be incremented/decremented from client's applet.

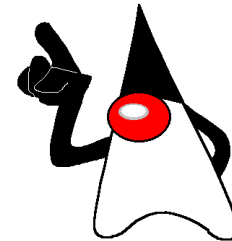
```
public interface RemoteCounter extends Remote {  
    public int getValue() throws RemoteException;  
    public void increment() throws RemoteException;  
    public void decrement() throws RemoteException;  
}
```



Remote Counter Implementation

```
public class RemoteCounterImp extends UnicastRemoteObject
implements RemoteCounter {

    protected int value;
    public RemoteCounterImp() throws RemoteException {
        super();
        value = 0;
    }
    public int getValue()
        { return value; }
    public void increment()
        { value++; }
    public void decrement()
        { value--; }
}
```



Stub and skeleton is generated.

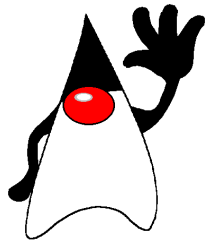
rmic RemoteCounterImp.class

Publishing the Counter

start rmiregistry or rmiregistry &

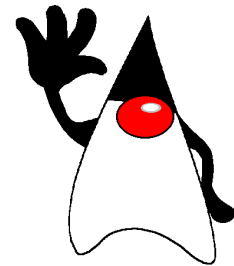
The RMI registry is a simple server-side bootstrap name server that allows remote clients to get a reference to a remote object.

```
public class RemoteCounterApp {  
    public static void main(String[] arg) {  
        try {  
            System.setSecurityManager(new RMISecurityManager());  
            RemoteCounterImp counter = new RemoteCounterImp();  
            Naming.rebind("//myhost:1099/COUNTER",counter);  
        }  
        catch (Exception e) {  
            System.out.println("Error while building remote counter!");  
        }  
    }  
}
```



Client of Remote Counter

```
public class RemoteCounterApplet extends Applet implements ActionListener {
    // Definition of GUI elements - buttons inc and dec, text field value, etc...
    protected RemoteCounter counter;
    public void init() {
        try {
            // Laying out of GUI components ...
            counter = (RemoteCounter) Naming.lookup("//"+getCodeBase().getHost()+"/COUNTER");
            updateText();
        } catch (Exception e) {}
    }
    public void actionPerformed(ActionEvent e) {
        try {
            if (e.getSource() == inc) counter.increment();
            if (e.getSource() == dec) counter.decrement();
            updateText();
        } catch (Exception e) {}
    }
    protected void updateText() throws RemoteException {
        value.setText(new Integer(counter.getValue()).toString());
    }
}
```



Behind the Scope ...

- ◆ **Internationalization**
- ◆ **Java Native Methods**
- ◆ **JAR - Java Archive**
- ◆ **Java JIT**
- ◆ **Java Run-Time Environment**
- ◆ **Java Foundation Classes**
- ◆ **Generating API documentation in HTML format from Java source code**